Predicate Logic

Anthony Narkawicz[1]

NASA Langley Formal Methods Group

anthony.narkawicz@nasa.gov

October 2012

[1] Based heavily on previous versions due to Paul Miner, Ben Di Vito, and Lee Pike

Quantification

- Quantified formulas are declared by quantifying free variables in the formula.
- Examples:

```
lem1: LEMMA FORALL (x: int, y: int): x * y = y * x

x, y, z: VAR int
lem2: LEMMA EXISTS z: x + z = 0
```

- Free variables in formulas are implicitly assumed to be universally quantified.

  Example:

```
lem3: LEMMA x * y = y * x
```

  is treated by the prover as

```
   |-
{1} FORALL (x: int, y: int): x + y = y + x
```

- *Skolemization* and *Instantiation* are used to eliminate quantifiers.

Skolemization

Suppose I want to prove:

If there exists a natural number m such that P(m) holds, then for all natural numbers n, Q(n) holds.

In PVS, this would look something like

```
{-1} EXISTS (m:nat): P(m)
    |-
{1} FORALL (n:nat): Q(n)
```

In mathematics, proof starts with “Let n be a natural number...”

That is just skolemization!!!

```
(skolem 1 ‘‘n’’)
```

Skolemization

```
{-1} EXISTS (m:nat): P(m)
    |-
{1} FORALL (n:nat): Q(n)
```

```
(skolem 1 ‘‘n’’)
```

This becomes

```
{-1} EXISTS (m:nat): P(m)
    |-
{1} Q(n)
```

In mathematics, the next step is “let m be a natural number such that P(m) holds”

This is skolemization too!!!

Skolemization

Skolemize both quantifiers in

```
{-1} EXISTS (m:nat): P(m)
    |-
{1} FORALL (n:nat): Q(n)
```

- Universal quantifiers in the consequent are skolemized.
- Existential quantifiers in the antecedent are skolemized.
- Skolemization is the process of introducing a fresh (i.e., unused in the sequent) constant (a *skolem constant*) to represent an arbitrary value in the domain.

Instantiation

Suppose I know

- *For all natural numbers m,* $P(m)$ *implies* $Q(m+1)$
- $P(19)$ *holds*

And I want to prove that

- *There exists a natural number n such that* $Q(n)$ *holds.*

In PVS, this is represented as

```
{-1} FORALL (m:nat): P(m) IMPLIES Q(m+1)
{-2} P(19)
    |-
{1} EXISTS (n:nat): Q(n)
```

Instantiation

```
{-1} FORALL (m:nat): P(m) IMPLIES Q(m+1)
{-2} P(19)
    |-
{1} EXISTS (n:nat): Q(n)
```

In mathematics, the first step is to say "Let m = 19 in formula -1".

This is instantiation

```
(inst -1 ``19'')
```

This substitutes 19 for m in that formula:

```
{-1} P(19) IMPLIES Q(20)
{-2} P(19)
    |-
{1} EXISTS (n:nat): Q(n)
```

Instantiation

```
{-1} P(19) IMPLIES Q(20)
{-2} P(19)
    |-
{1} EXISTS (n:nat): Q(n)
```

The next step in math is to say "let n = 20 in formula 1".

This is instantiation too!!!

```
(inst 1 ``20'')
```

This becomes

```
{-1} P(19) IMPLIES Q(20)
{-2} P(19)
    |-
{1} Q(20)
```

Prove this using (assert)

Instantiation

Instantiate both quantifiers in

```
{-1} FORALL (m:nat): P(m) IMPLIES Q(m+1)
{-2} P(19)
    |-
{1} EXISTS (n:nat): Q(n)
```

- Universal quantifiers in the antecedent are instantiated.
- Existential quantifiers in the consequent are instantiated.
- Instantiation is the process of replacing a quantified variable with a previously-declared constant.

Universal vs. Existential Variables

Location	Top-level quantifier	
	`FORALL`	`EXISTS`
Antecedent	`(inst)`	`(skolem)`
Consequent	`(skolem)`	`(inst)`

- Embedded quantifiers must be brought to the outermost level for quantifier rules to apply.
  - E.G. You can't instantiate the quantifier in
    `{-1} P(10) IMPLIES (FORALL (m:nat): P(m) IMPLIES Q(m+1))`
- `skolem` and `inst` each have options.
- Simple versions of these are automated in PVS.

Skolemization in PVS

- Skolem constants are generated with explicit commands.
- There is a `skolem` command and several variants.
- Syntax: `(skolem! &optional (fnums *) ...)`
- A common variant is (`skosimp*`) which applies (`skolem!`) and (`flatten`)
- Syntax: `(skosimp* &optional preds?)`
- Generates Skolem constants for formulas given in `fnums`
- Only top-level quantifiers may be skolemized.
- Usually invoked without arguments, applying it to the whole sequent.
- The Emacs command `M-x show-skolem-constants` shows the currently active constants in a separate emacs buffer.

Practical Skolemization

Commands to use:

1. (`skolem` -1 "k")
   - introduces the constant k in place of a quantified variable in formula -1
2. (`skolem!`)
   - skolemizes every quantifier that can be skolemized and introduces its own constants
   - Usually quantified variable `x` becomes the constant `x!1` or `x!2`...
3. (`skosimp*`)
   - applies (`skolem!`) (`flatten`)
   - Often used at the start of a proof to get to the point where you really want to start
4. (`skeep`)
   - skolemize and "keep" variable names
   - variable `x` becomes constant `x` instead of `x!1`

Practical Skolemization

How I typically use these commands (verbatim):

- (skeep) 40% of the time
- (skosimp*) 40% of the time
- (skolem -1 "k") 20% of the time

I could probably use (skosimp*) 95% of the time

Moral of the story: skolemization in PVS is pretty simple

Instantiating Quantifiers

- Instantiation involves substituting suitable terms for quantifiers in the current sequent.
- Basic syntax: `(inst fnum &rest terms)`
- Typechecking is performed on the terms.
  - You can't instantiate `(FORALL (d:Dog): loud?(d))` with `c:Cat`
  - This can generate new branches in the proof: *PVS may require you to prove that c (cat) is a dog*
- Several variants of `inst`
  - `(inst -1 ''3'')` instatiates quantifier in formula -1 with 3
  - `(inst-cp -1 ''3'')` instantiates quantifier in formula -1 with 3 but also keeps a copy of the original formula
  - `(inst?)` PVS guesses which instantiation you want and the formula number
  - `(inst? -3)` PVS guesses which instantiation you want in formula -3

Instantiate & Copy

- Syntax: `(inst-cp fnum &rest terms)`
- Works just like `inst`, but saves a copy of the formula in quantified form
- This is useful if you want to use a lemma twice.
- One instance may need one term for the instantiation of a variable, while another instance may need a different term, so . . .
- . . . `inst-cp` allows you to have it both ways.

Find my Constant

- Syntax: `(inst? &optional (fnums *) ...)`
- Similar to `inst`, but tries to automatically find the terms for substitution
- This is useful in most proof situations.
- There are usually expressions lying around in the sequent that are the terms you want to substitute.
- `inst?` is pretty good at finding them.
- The larger the sequent, however, the more candidate terms exist to choose from, causing the success rate to drop.

PVS Theory for Examples

We will be using a simple PVS theory to illustrate basic prover commands:

```
%%%     Examples and exercises for basic prover commands

pred_basic: THEORY
BEGIN

arb: TYPE+                          % Arbitrary nonempty type

arb_pred: TYPE = [arb -> bool]      % Predicate type for arb

a,b,c: arb                          % Constants of type arb

x,y,z: VAR arb                      % Variables of type arb

P,Q,R: arb_pred                     % Predicate names

          ⋮
```

Sample Quantified Formulas

```
          ⋮

quant_0: LEMMA  (FORALL x: P(x)) => P(a)

quant_1: LEMMA  (FORALL x: P(x)) => (EXISTS y: P(y))

quant_2: LEMMA  (EXISTS x: P(x)) OR (EXISTS x: Q(x))
                  IFF (EXISTS x: P(x) OR Q(x))

l,m,n:  VAR int

distrib: LEMMA  l * (m + n) = (l * m) + (l * n)

END pred_basic
```

Skolem Constants (Cont'd)

Starting proof of formula `distrib` from theory `prover_basic`:

```
distrib :

  |-------
{1}    FORALL (l: int, m: int, n: int):
           l * (m + n) = (l * m) + (l * n)

Rule? (skolem!)
Skolemizing,
this simplifies to:
distrib :

  |-------
{1}    l!1 * (m!1 + n!1) = (l!1 * m!1) + (l!1 * n!1)
```

The variables `l`, `m`, `n` have been replaced with the skolem constants `l!1`, `m!1`, `n!1`.

Example of Instantiation

```
quant_0 :

  |-------
{1}    (FORALL x: P(x)) => P(a)

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
quant_0 :

{-1}    (FORALL x: P(x))
  |-------
{1}    P(a)

Rule? (inst -1 "a")
Instantiating the top quantifier in -1 with the terms: a,
Q.E.D.
```

Another Example of Instantiation

Try getting the prover to automatically find the instantiation.

```
quant_1 :

  |-------
{1}    ((FORALL x: P(x) => Q(x)) AND P(a)) => Q(a)

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
quant_1 :

{-1}    (FORALL x: P(x) => Q(x))
{-2}    P(a)
  |-------
{1}    Q(a)
```

Looks like the constant “a” is what we want.

Another Instantiation Example (Cont'd)

```
Rule? (inst?)
Found substitution:
x gets a,
Instantiating quantified variables,
this simplifies to:
quant_1 :

{-1}    P(a) => Q(a)
[-2]    P(a)
  |-------
[1]    Q(a)

Rule? (prop)
Applying propositional simplification,
Q.E.D.
```

The prover made the right pick!

Can the Prover Always Find an Instantiation?

```
quant_2 :

  |-------
{1}    (FORALL x: P(x)) => (EXISTS y: P(y))

Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
quant_2 :

{-1}    (FORALL x: P(x))
  |-------
{1}    (EXISTS y: P(y))
```

What will INST? do here?

Find an Instantiation? (Cont'd)

```
Rule? (inst?)
Couldn't find a suitable instantiation for any
quantified  formula.  Please provide partial instantiation.
No change on: (INST?)
quant_2 :

{-1}    (FORALL x: P(x))
  |-------
{1}    (EXISTS y: P(y))
```

The prover gives up — it can't do the "creative" work of finding a viable term if it's not present in the sequent.

Find an Instantiation? (Cont'd)

```
Rule? (inst + "a")
Instantiating the top quantifier in + with the terms:
 a,
this simplifies to:
quant_2 :

[-1]    (FORALL x: P(x))
  |-------
{1}    P(a)

Rule? (inst?)
Found substitution:
x gets a,
Instantiating quantified variables,
Q.E.D.
```

Need to supply your own term in this case.

Hiding Formulas

Two commands tell the prover to temporarily forget information and then recall it later.
The first tells the prover which items to ignore

- Syntax: `(hide &rest fnums)`.
- Causes the designated formulas to be hidden away.
- Those formulas will not be used in making deductions.
- This is useful if you have a complicated sequent and some of the formulas look irrelevant.
- Also useful if a formula has already served its purpose.
- Saves processing time during proof steps.

Revealing Formulas

The second command allows you to bring hidden formulas back

- Syntax: `(reveal &rest fnums)`
- Restores the designated formulas to the current sequent
- Makes the deletion of information through the `hide` command safe
- The Emacs command `M-x show-hidden-formulas` tells you what is hidden and what their current formula numbers are.

Decision Procedures

PVS uses decision procedures to supplement logical reasoning.

- Terminating algorithms that can decide whether a logical formula is valid or invalid
- These constitute *automated theorem-proving*, so they usually provide no derivations.

  Example: a truth table for propositional logic
- PVS integrates a number of decision procedures including
  - Theory of equality with uninterpreted functions
  - Linear arithmetic over natural numbers and reals
  - PVS-specific language features such as function overrides

Various prover rules apply decision procedures in combination with other reasoning techniques.

- Important feature for achieving automation
- At the cost of visibility into intermediate steps

Deductive Hammers: Small To Large

The prover has a hierarchy of increasingly muscular simplification rules.

`PROP`	Repeated application of `flatten` and `split`
`BDDSIMP`	Propositional simplification using Binary Decision Diagrams (BDDs)
`ASSERT`	Applies type-appropriate decision procedures and auto-rewrites
`GROUND`	Propositional simplification plus decision procedures
`SMASH`	Repeatedly tries `BDDSIMP`, `ASSERT`, and `LIFT-IF`
`GRIND`	All of the above plus definition expansion and `INST?`

Automated Deduction Tips

- Typically, these simplification rules are invoked without arguments.
- Examples: `(assert)`, `(ground)`, `(grind)`
- Caution: `GRIND` is fairly aggressive
  - Can take a while to complete
  - Might leave you in a strange place when it's done
  - Might need to be interrupted to abort runaway behavior

Using Type Information

The prover needs to be asked to reveal information about typed expressions

- A command for importing type predicate constraints:
  - Syntax: `(typepred &rest exprs)`
  - Causes type constraints for expressions to be added to sequent
  - Subtype predicates are often recalled this way

Type-Predicate Example

```
bounded1 :

  |-------
{1}  FORALL (a: {x: real | abs(x) < 1}):
         a * a < 1

Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
bounded1 :

  |-------
{1}   a!1 * a!1 < 1

Rule? (typepred "a!1")
Adding type constraints for  a!1,
this simplifies to:
bounded1 :

{-1}  abs(a!1) < 1
  |-------
[1]   a!1 * a!1 < 1
```

Summary

- A constant companion:
  `skolem` universals in the consequent & existentials in the antecedent.
- For one and all:
  `inst` universals in the antecedent & existentials in the consequent.
- Hide 'n Seek: `hide` & `reveal`
- Automatic for the provers:
  `prop`, `assert`, `ground`, `grind`.
- Hey formula, what's your type?
  `typepred` & `typepred!`